RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 4: Java Collection Framework and Generics/Parameterized Typing

Lecture Slides

Java Collections

- An Object that groups multiple elements into a single unit
- To store, retrieve and manipulate data, and to transmit data from one method to another
- Typically represents data items belonging to a natural group, such as Library Catalogue (a collection of library materials and borrow-status)

JAVA Collection Implementation in earlier versions (still supported)

1. Vector - collection of Java Objects (contents can be instantiated from different classes) without prior knowledge of the size
2. Hashtable – collection of name-value pairs, like a dictionary, easy lookup
3. Array – collection of objects of same class or same primitive datatypes with known size

Java Collection Framework (JCF)

A 'Unified Architecture' for representing and manipulating collections

Collection Framework contains:

• **Interfaces**: abstract data types representing collections. Interfaces allow collections to be manipulated independently of the details of their representation.

`java.util.Collection etc.`

• **Implementations**: concrete implementations of the collection interfaces. In essence, these are *reusable data structures*.

`java.util.ArrayList etc.`

• **Algorithms**: methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces. These algorithms are said to be *polymorphic* because the same method can be used on many different implementations of the appropriate collections interface. In essence, algorithms are *reusable functionality*.

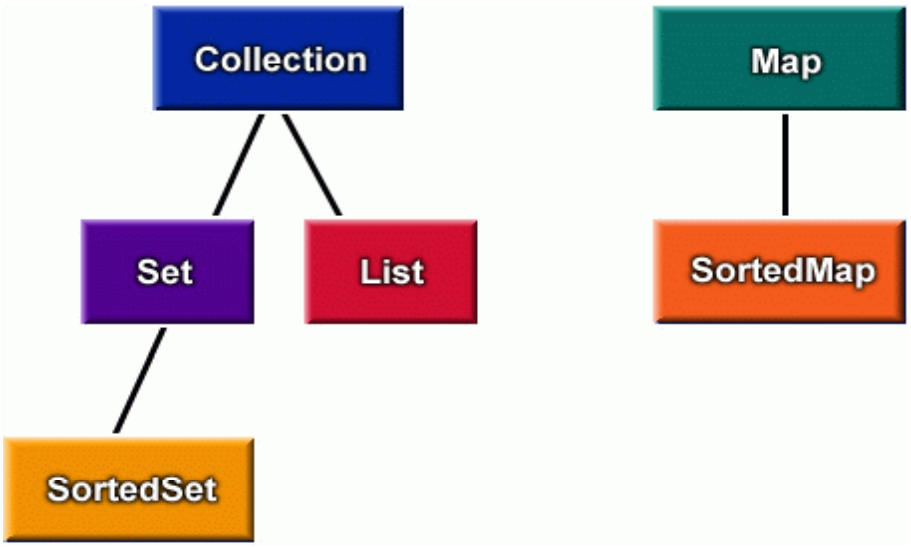
`java.util.Collections` (note the 's' on the end!)

One of the basic features of OO Programming is *reusability*.

Pros and Cons of JCF

Pros	Cons
1. Reduces Programming Effort (?)	1. Complexity (?)
2. Increases Speed and Quality (?)	
3. Allows interoperability among unrelated code	
4. Reduces learning effort (?)	
5. Reduces design effort	
6. Fosters software reuse	

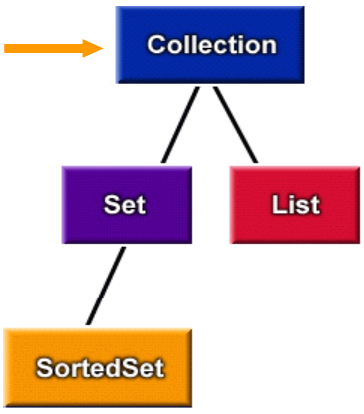
Collection Interface Hierarchy



No unifying interface but can retrieve a Collection from a Map with Map.values()

Modification operations in each interfaces are designated *optional*: a given implementation may not support some of these operations. If an unsupported operation is invoked, a collection throws an UnsupportedOperationException

Interface : Collection



What is Iterator?

A Collection represents a group of objects, known as its *elements*. The primary use of the Collection interface is to pass around collections of objects where maximum generality is desired.

```
public interface Collection
{
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);    // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);    // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear();                    // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

Iterator

Iterator Interface (new)

Iterator allows the caller to remove elements from the underlying collection during the iteration with well-defined semantics.

```
public interface Iterator
{
    boolean hasNext();
    Object next();
    void remove();    // Optional
}
```

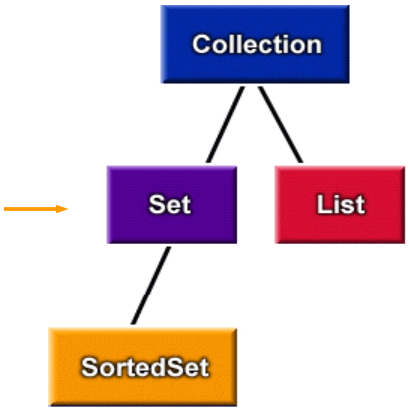
Enumeration Interface (old)

An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the nextElement method return successive elements of the series. Eg, StringTokenizer

Two Methods:

- `boolean hasMoreElements()`
- `Object nextElement()`

Interface : Set



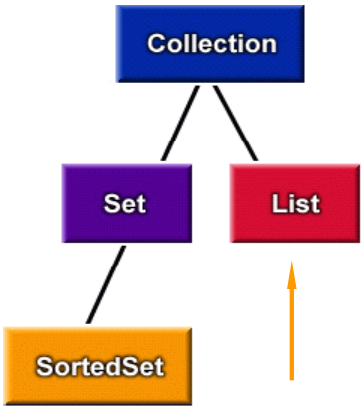
- A **Set** is a Collection that cannot contain duplicate elements.
- Set models the mathematical *set* abstraction.
- The Set interface extends Collection and contains *no* methods other than those inherited from Collection.
- Set also adds a stronger contract on the behavior of the equals and hashCode operations (inherited from Object), allowing Set objects with different implementation types to be compared meaningfully. Two Set objects are equal if they contain the same elements.

JDK Standard Implementation
HashSet – best performing and stored in hash table.
TreeSet - guarantees ordering and stored in red-black tree

What are the common Set Operations?

Subset	`s1.containsAll(s2)`
Union	`s1.addAll(s2)`
Intersection	`s1.retainAll(s2)`
Set Difference	`s1.removeAll(s2)`

Interface: List



A **List** is an ordered Collection (sometimes called a *sequence*). Lists may contain duplicate elements.

In addition to the operations inherited from Collection, the List interface includes operations for:

1. **Positional Access (indexed)**: manipulate elements based on their numerical position in the list.
2. **Search**: search for a specified object in the list and return its numerical position.
3. **List Iteration**: extend Iterator semantics to take advantage of the list's sequential nature.
4. **Range-view**: perform arbitrary *range operations* on the list.

JDK Standard Implementation

ArrayList – best performing and stored in hash table.
LinkedList - better performance under certain circumstances
Vector – retrofitted to implement List

Interface: List (Continued…)

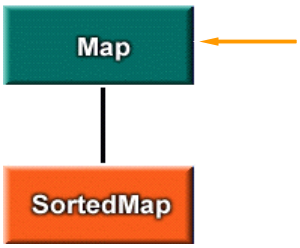
```
public interface List extends Collection {
   // Positional Access
   Object get(int index);
   Object set(int index, Object element); // Optional
   void add(int index, Object element);   // Optional
   Object remove(int index);              // Optional
   abstract boolean addAll(int index,
                           Collection c); // Optional

   // Search
   int indexOf(Object o);
   int lastIndexOf(Object o);

   // Iteration
   ListIterator listIterator();
   ListIterator listIterator(int index);

   // Range-view
   List subList(int from, int to);
}
```

```
public interface ListIterator
   extends Iterator {

   boolean hasNext();
   Object next();

   boolean hasPrevious();
   Object previous();

   int nextIndex();
   int previousIndex();

   void remove();          // Optional
   void set(Object o);     // Optional
   void add(Object o);     // Optional
}
```

Most of the polymorphic algorithms in the **JCF** classes (e.g. java.util.Collections) apply specifically to List not Collection.

- `sort(List)`
- `shuffle(List)`
- `reverse(List)`
- `fill(List, Object)`
- `copy(List dest, List src)`
- `binarySearch(List, Object)`

Interface: Map



A Map is a collection that maps keys to values.

A map cannot contain duplicate keys: Each key can map to at most one value.

JDK Standard Implementation
HashMap– best performing and stored in hash table.
TreeMap - guarantees ordering and stored in red-black tree
Hashtable – retrofitted to implement Map

```
public interface Map
{
    // Basic Operations
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk Operations
    void putAll(Map t);
    void clear();

    // Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interface for entrySet elements
    public interface Entry
    {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```


Comparable and Comparator Interface

If the list consists of String elements, it will be sorted into lexicographic (alphabetical) order. If it consists of Date elements, it will be sorted into chronological order. How?

Both String and Date implements the ***Comparable*** interface that provides a ***natural ordering*** for a class, which allows objects of that class to be sorted automatically.

- a list whose elements do not implement Comparable, Collections.sort(list) will throw a ClassCastException.

- a list whose elements cannot be compared *to one another*, Collections.sort will throw a ClassCastException.

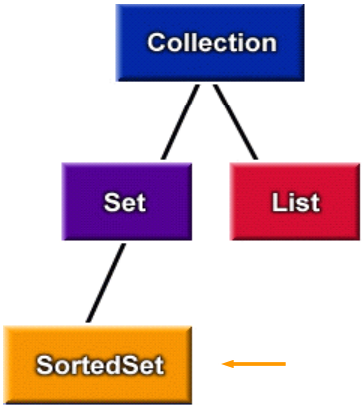
- Elements that can be compared to one another are called *mutually comparable*. While it is possible to have elements of different types be mutually comparable, none of the JDK types (Byte, Character, Long, Integer, Short, Double, Float, String, Date) permit inter-class comparison.

```
public interface Comparable
{
   public int compareTo(Object o);
}
```

```
public interface Comparator
{
   int compare(Object o1, Object o2);
}
```

Interface: SortedSet

A **SortedSet** is a Set that maintains its elements in ascending order, sorted according to the elements' ***natural order***, or according to a Comparator provided at SortedSet creation time.



In addition to the normal Set operations, the SortedSet interface provides operations for:

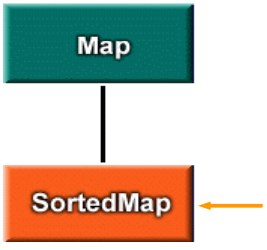
Range-view: Performs arbitrary *range operations* on the sorted set.
Endpoints: Returns the first or last element in the sorted set.
Comparator access: Returns the Comparator used to sort the set

```
public interface SortedSet extends Set {
   // Range-view
   SortedSet subSet(Object fromElement, Object toElement);
   SortedSet headSet(Object toElement);
   SortedSet tailSet(Object fromElement);

   // Endpoints
   Object first();
   Object last();

   // Comparator access
   Comparator comparator();
}
```


Interface: SortedMap

A **SortedMap** is a Map that maintains its elements in ascending order, sorted according to the elements' ***natural order***, or according to a Comparator provided at SortedMap creation time.

In addition to the normal Map operations, the SortedMap interface provides operations for:



Range-view: Performs arbitrary *range operations* on the sorted map.
Endpoints: Returns the first or last key in the sorted map.
Comparator access: Returns the Comparator used to sort the map

```
public interface SortedMap extends Map {
    // Range-view
    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap headMap(Object toKey);
    SortedMap tailMap(Object fromKey);

    // Endpoints
    Object first();
    Object last();

   // Comparator access
    Comparator comparator();

}
```

Implementation

Implementations are the actual data objects used to store collections, which implement the *core collection interfaces*

General-purpose Implementations

General-purpose implementations are the public classes that provide the primary implementations of the core collection interfaces. e.g. ArrayList, HashMap

Convenience Implementations

Convenience implementations are mini-implementations, typically made available via *static factory methods* that provide convenient, efficient alternatives to the general-purpose implementations for special collections e.g. Collections.singletonList().

Wrapper Implementations

Wrapper implementations are used in combination with other implementations (often the general-purpose implementations) to provide added functionality. e.g. Collections.unmodifiableCollection();

General Purpose Implementations

Two implementations for each interface except Collection has been provided.

		Implementations			
		Hash Table	***Resizable Array***	***Balanced Tree***	***Linked List***
Interfaces	***Set***	***HashSet***		***TreeSet***	
	List		***ArrayList, Vector***		***LinkedList***
	Map	***HashMap***		***TreeMap***	

(Reference : Java Tutorial – Joshua Bloch, collection trail)

An Iterator Example

```
c1 = new LinkedList();
// Process the linked list
Iterator iter = c1.iterator();
while (iter.hasNext()) {
   Object obj = iter.next();
   //do what you want to obj
}
c2 = new HashSet();
// Process the hash set
iter = c2.iterator();
while (iter.hasNext()) {
   Object obj = iter.next();
   //do what you want to obj
}
```

Note identical code for generating the sequence of objects

typically, we must cast the object to use it properly

Java 1.5: Use of for-each loop instead of an Iterator

```
c1 = new LinkedList();
// Process the linked list
Iterator iter = c1.iterator();
while (iter.hasNext()) {
   Object obj1 = iter.next();
   //do what you want to obj1
}
c2 = new HashSet();
// Process the hash set
iter = c2.iterator();
while (iter.hasNext()) {
   Object obj2 = iter.next();
   //do what you want to obj2
}
```

```
c1 = new LinkedList();
// Process the linked list
for(Object obj1: c1)
   //do what you want to obj1
}

c2 = new HashSet();
// Process the hash set
for(Object obj2: c2) {
   //do what you want to obj2
}
```

Use of Generic classes

- Java 5 allows the use of Generic classes and methods
- Traditionally Java programmers have used inheritance & polymorphism to create flexible classes.
- A flexible Stack class can be written to store any object by creating an array of Object references
- However it does not prevent the wrong type of object being added to the stack at compile time – resulting in a runtime error at a later time.
- It also requires casting when an object is retrieved.

```
Class Stack
{
   private Object elems[];
   void push(Object o) {…}
   Object pop() {…}
}

Stack custStack = new Stack(10);

custStack.push(new Customer(…));
custStack.push(new Customer(…));
custStack.push(new Account(…));
…
Cutomer c = (Customer)
custStack.pop();
…
```


Using JCF Generic classes

- All the JCF classes we have seen before are created to be used as generic classes
- For example, an ArrayList instance can be created to store only Account objects by passing the type (Account) to the constructor and the reference as in:

```
List<Account> accList = new ArrayList<Account>();
```

- Similarly to map a customer name (String) to Account objects we can use:

```
Map<String,Account> hashMap = new
   HashMap<String,Account>();
```

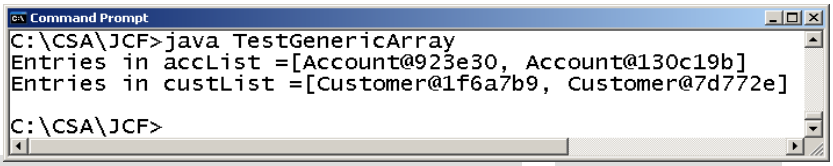
- The same classes when used without specifying any type information reduces to raw type which is equivalent to:

```
List<Object> accList = new ArrayList<Object>();
Map<Object,Object> hashMap = new
   HashMap<Object,Object>();
```

Example using ArrayList

```
import java.util.*;
class Account {  }
class Customer { }

public class TestGenericArray {
   public static void main(String[] args) {
      List<Account> accList = new ArrayList<Account>();
      accList.add( new Account() );
      accList.add( new Account() );

      List<Customer> custList = new ArrayList<Customer>();
      custList.add( new Customer() );
      custList.add( new Customer() );
//    custList.add( new Account() );

      System.out.println("Entries in accList =" + accList);
      System.out.println("Entries in custList =" + custList);

   }
}
```

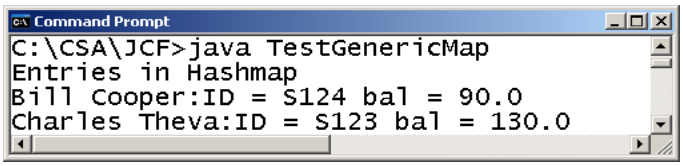
compiler detects type mismatch



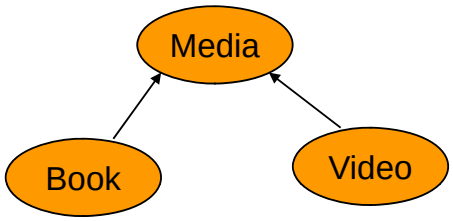
Example using HashMap

```
import java.util.*;
Customer Account { …
public class TestGenericMap {
  public static void main(String[] args) {
  HashMap<String,Account> hashMap =
                  new HashMap<String,Account>();
      hashMap.put("Charles Theva", new Account("S123",130.0));
      hashMap.put("Bill Cooper", new Account("S124",90.0));
//    hashMap.put(1234, new Account("S126",220.0));
      System.out.println("Entries in Hashmap");
      displayMap(hashMap);
   }
   public static void displayMap(Map<Integer, Account> m)
   {
      Set<Integer> keySet = m.keySet();
      Iterator<Integer> iterator = keySet.iterator();
      while (iterator.hasNext())
      {
         Integer key = iterator.next();
         System.out.println(key + ":“
                 + m.get(key));
      }
   }
}
```

Compiler detects type mismatch



Consider a simple library system




```
import java.util.*;
class Library {
   private List resources =
             new ArrayList();
   public void add(Media x) {
      resources.add(x);
   }
   public Media retrieveLast() {
      int size = resources.size();
      if (size > 0) {
         return (Media)
           resources.get(size-1);
      }
      return null;
   }
}
```

```
public class TestLibrary
{  public static void main(String  args[])
   {  Library myBooks =  new Library();
      myBooks.add(new Book());
      myBooks.add(new Book());
//    myBooks.add(new Video());
      Book lastBook = (Book) myBooks.retrieveLast();
      lastBook.print();
   }
}
```

For storing Book objects

Cannot detect at compile time resulting in runtime error when retrieving

Explicit cast needed

A Parameterized Library class

The generic library class has a single type parameter E, allowing it to store objects of type E.

Using parameterized type E

Uses the services of parameterized ArrayList reference

Constructor

Allows any object of type E to be added

Objects retrieved are of type E

```
import java.util.*;

class Library<E>
{
   private List<E> resources = new ArrayList<E>();

   public void add(E x)
   {
      resources.add(x);
   }

   public E retrieveLast() {
      int size = resources.size();
      if (size > 0) {
         return resources.get(size-1);
      }
      return null;
   }
}
```

Using the Parameterized Library

- When using the parameterized (generic) Library class a type must be passes to the type parameter E.
- Note the element extracted from the parameterized Library need not be cast.

```
public class TestLibrary {
   public static void main(String args[]) {
      Library<Book> myBooks =  new Library<Book>();
      myBooks.add(new Book());
      myBooks.add(new Book());
      Book lastBook = myBooks.retrieveLast();
      lastBook.print();

      Library<Video> myVideos =  new Library<Video>();
      myVideos.add(new Video());
      myVideos.add(new Video());
      myVideos.add(new Video());
      Video lastVideo = myVideos.retrieveLast();
      lastVideo.print();
   }
}
```

Creating a Library to store Book objects

Creating a Library to store Video objects

No casting needed

Things to note when creating and using a Parameterized classes

- Primitive types cannot be passed as parameters.

```
List<Integer> numbers = new ArrayList<Integer>();
List<int> numbers = new ArrayList<int>();
```

- When a class uses parameterized type T, the type parameter T can be used as a reference but not for constructing.

```
T object = …
T[] a = …
         = new T();
         = new T[10];
```

- Though the parameter cannot be used as a constructor it can be used for casting

```
E e2=(E) new Object();
E[] e3 = (E[])new Object[10];
```

- Generic classes cannot be array base type (but can be a parameterized collection)

```
Library<Video> videoLibs[] =  new Library<Video>[10];
List<Video> vidLibs[] =  new ArrayList<Video>[10];
```

See http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf for more info.

A parameterized Stack class (without using other parameterized classes)

- A stack class that allows any objects of the same type to be stacked.
- It will initially create an array with initial capacity to store 10 elements – the capacity will be increased when necessary.
- Internally it creates an array of Object references and casts it to the parametric type as in:

```
private T[] elements;
…
public GStack(int capacity) {
    elements = (T[])new Object[capacity];
 }
```

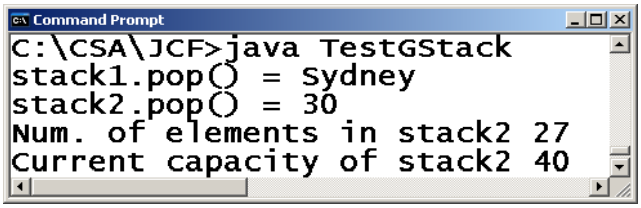
- Whenever capacity is increased it creates a new block of memory and copies the existing elements.


```
/* Adapted from Supplement Q: Generic Types By Y. Daniel Liang */
class GStack<T> {
   private T[] elements;
   private int size;

   // constructs a stack with the default capacity 10
   public GStack() { this(10); }

   // constructs a stack with the specified initial capacity
   public GStack(int capacity) {
      elements = (T[])new Object[capacity];
   }

   // puts the new element into the top of stack
   public T push(T value) {
      if (size >= elements.length)
         setCapacity(elements.length * 2);
      return elements[size++] = value;
   }

   // returns and removes the top element from the stack
   public T pop() {
      return elements[--size];
   }
   // returns top element without removing
   public T peek() {
      return elements[size - 1];
   }
```

```
   // tests whether the stack is empty
   public boolean empty() {
      return size == 0;
   }

   // returns the number of elements in the stack
   public int getSize() {
      return size;
   }

   // returns the capacity
   public int getCapacity() {
      return elements.length;
   }

   // sets new capacity - must be greater than current capacity
   public boolean setCapacity(int newCapacity) {
      if ( newCapacity > elements.length ) {
         T[] temp = (T[])new Object[elements.length * 2];
         System.arraycopy(elements, 0, temp, 0, elements.length);
         elements = temp;
         return true;
      }
      else
         return false;
   }
}
```


Using the generic Stack

```
public class TestGStack
{  public static void main(String args[])
   {  GStack<String> stack1 = new GStack<String>();
      stack1.push("Perth");
      stack1.push("Melbourne");
      stack1.push("Sydney");
      System.out.println("stack1.pop() = " + stack1.pop());

      GStack<Integer> stack2 = new GStack<Integer>();
      stack2.push(10);
      stack2.push(20);
      stack2.push(30);
      System.out.println("stack2.pop() = " + stack2.pop());
      for (int i=0; i<25; i++)
         stack2.push(i);
      System.out.println("Num. of elements in stack2 " +
                                              stack2.getSize());
      System.out.println("Current capacity of stack2 " +
                                              stack2.getCapacity());
   }
}
```



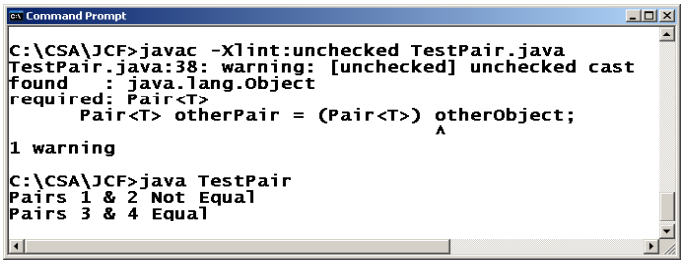
Comparison of Parametric classes

- The next program shows how a generic class could provide a method to compare two objects for equality. The Pair class keeps a pair of objects of the same type. It takes that type as a parameter.
- The equals method takes an Object reference to the other object. The Object reference is cast to the type of the current object using:
  - `Pair<T> otherPair = (Pair<T>) otherObject;`
- A cast such as this generates compiler warnings. To see the details of warnings compile with
  - `javac –Xlint:unchecked TestPair.java`
- The equals method first verifies it is an objects of the same class before comparing the parts that make up the two objects.
- See source code for better example of equals()


```
class Pair<T>
{
   private T first;
   private T second;
   public Pair()
   {
      first = null;
      second = null;
   }
   public Pair(T first, T second)
   {
      this.first = first;
      this.second = second;
   }
   public boolean equals(Object otherObject)
   {
      if (otherObject == null) return false;
      if (getClass() != otherObject.getClass())
         return false;
      Pair<T> otherPair = (Pair<T>) otherObject;
      return (first.equals(otherPair.first)
                && second.equals(otherPair.second));
   }
}
```

Testing the Pair class

```
public class TestPair
{  public static void main(String args[])
   {
      Pair<String> pair1 = new Pair<String>("10+5", "20+5");
      Pair<String> pair2 = new Pair<String>("15","25");
      if (pair1.equals(pair2))
         System.out.println("Pars 1 & 2 Equal");
      else System.out.println("Pairs 1 & 2 Not Equal");
      Pair<Integer> pair3 = new Pair<Integer>(10+5, 20+5);
      Pair<Integer> pair4 = new Pair<Integer>(15,25);
      if (pair3.equals(pair4))
         System.out.println("Pairs 3 & 4 Equal");
      else System.out.println("Pairs 3 & 4 Not Equal");
   }
}
```



Bounds for type parameters

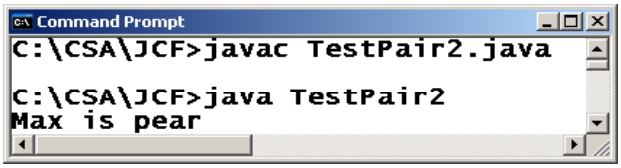
- Parametric classes may not make sense for all possible types
- In the parameterized library class we may want restrict the parameter type to Media or its subclasses.
- We may expect all items in the library to have a catalogue number or a method to get the expiry date.
- We can specify bounds for the type of parameters by using extends or implements clause (or both) as in:
  - `class Library<E extends Media> { …`
- As another example we extend the Pair class by incorporating a method named max() to return the bigger of its two objects.
- Java provides the Comparable<T> interface with a method public int compareTo(T other) that returns zero, negative or positive value depending on which object is larger.
- The Pair objects can be compared using this method if we specify bounds for the Parametric Pair class.
  - `class Pair<T extends Comparable<T>>`

Pair class with extends bound

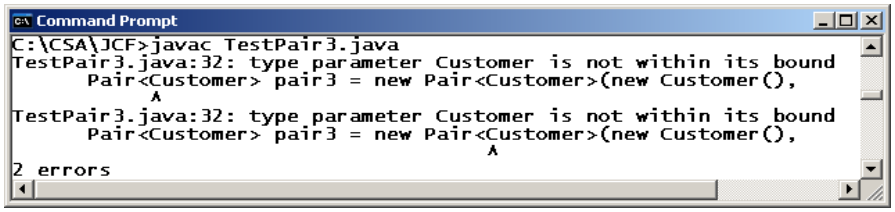
```
class Pair2<T extends Comparable<T>> {
   private T first;
   private T second;

   public Pair() {
      first = null;
      second = null;
   }
   public Pair(T first, T second) {
      this.first = first;
      this.second = second;
   }
   public T max() {
      if (first.compareTo(second) >= 0)
         return first;
      else
         return second;
   }
}
```


Testing the Bounded Pair class

```
public class TestPair2 {
  public static void main(String args[]) {
     Pair2<String> pair1 = new Pair2<String>("apple", "pear");
     System.out.println("Max is " +  pair1.max());
   }
}

class Customer {}

public class TestPair3 {
  public static void main(String args[]) {
      Pair2<Customer> pair3 = new Pair2<Customer>(new Customer(),
                              new Customer());
      System.out.println("Max is " +  pair3.max());
   }
}
```



Customer does not implement Comparable



A class with multiple type parameters

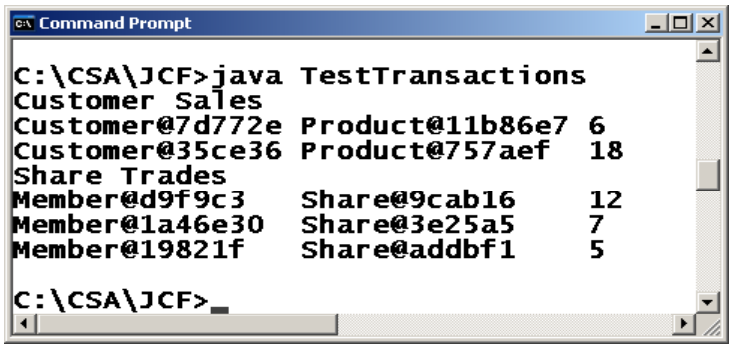
- As an example for a class with multiple type parameters we have created a Transaction class which associate one type of object with another type (Customer and Product or Member and Share) together with another to represent the quantity.
- Internally it maintains 3 arrays and a String to store the title to used in printing. The type of first two arrays depends on the type parameters passed. The third one is an Integer array to keep the quantity of transactions (share trades, sales …)

```
class Transactions<T1, T2> {
   private List<T1> owners = new ArrayList<T1>();
   private List<T2> items = new ArrayList<T2>();
   private List<Integer> nums= new
      ArrayList<Integer>();
   private String title;
   public Transactions(String title) { this.title = title; }
}
```



```
import java.util.*;
class Transactions<T1, T2>
{
   private List<T1> owners = new ArrayList<T1>();
   private List<T2> items = new ArrayList<T2>();
   private List<Integer> nums= new ArrayList<Integer>();
   private String title;
   public Transactions(String title) { this.title = title; }

   public void add(T1 owner, T2 item, int num)
   {
      owners.add(owner);
      items.add(item);
      nums.add(num);
   }
   public void list()
   {
      System.out.println(title);
      for (int i=0; i<owners.size(); i++)
        System.out.println(owners.get(i)+"\t"
           +items.get(i)+"\t"+nums.get(i));
   }
}
```

```
class Customer2 { }
class Product { }
class Member { }
class Share { }
public class TestTransactions {
  public static void main(String args[]) {
      Transactions<Customer2,Product> sales
            = new Transactions<Customer2,Product>("Customer Sales");
      Transactions<Member,Share> trades
            = new Transactions<Member,Share>("Share Trades");
      sales.add(new Customer2(), new Product(), 6);
      sales.add(new Customer2(), new Product(), 18);
      trades.add(new Member(), new Share(), 12);
      trades.add(new Member(), new Share(), 7);
      trades.add(new Member(), new Share(), 5);
      sales.list();
      trades.list();
   }
}
```



Generic Methods

- Generic methods can be member of generic classes or normal (ordinary) classes
- In the next example we have created a utility class that has three generic methods all declared as static.
- The method getMid() takes an array of elements of any type and returns the middle one.
- The method getLast() returns the last one.
- The print() method takes an array of elements of any type and prints them in a row.

```
class GenericMethods
{
   public static <T> T getMid(T[] a)
   {
      return a[a.length/2];
   }
   public static <T> T getLast(T[] a)
   {
      return a[a.length-1];
   }
   public static <T> void print(T[] a)
   {
      for (int i=0; i<a.length; i++)
         System.out.print(a[i] + " ");
      System.out.println();
   }
}
```



```
public class TestGenericMethods {
   public static void main(String args[]) {
      String s[] = { "Apples", "Oranges", "Grapes" };
      Integer nums[] = { 30 , 40 , 50, 90 , 80};

      String midS = GenMethods.<String>getMid(s);
      Integer midN = GenMethods.<Integer>getMid(nums);
      String lastS = GenMethods.<String>getLast(s);
      Integer lastNum = GenMethods.<Integer>getLast(nums);

      System.out.println("Mid String = " + midS);
      System.out.println("Mid Number = " + midN);
      System.out.println("Last String = " + lastS);
      System.out.println("Last Number = " + lastNum);
      System.out.println("** Testing Generic print **");
      GenMethods.<String>print(s);
      GenMethods.<Integer>print(nums);
   }
}
```